1. High-Level Ask

1.1 Objective

Build a **GPU-accelerated AprilTag detection engine** on Jetson Orin NX that can achieve **effective ~120 FPS** using 720p camera input for FRC-style localization, with performance substantially exceeding the current CPU-only implementation.

1.2 Scope

- Single camera initially (IMX477 / AR0234 @ 1280×720), scalable to multiple cameras later.
- Detection of **FRC AprilTags** (e.g. TAG_36H11) with:
    - Reliable detection up to typical FRC distances (3–8 m+),
    - Latency low enough to support fast swerve control and pose correction.
- GPU handles:
    - Raw → Gray conversion
    - Undistortion + decimation
    - Gradient / edge / quad candidate generation
    - Tag decode
    - PnP (3D pose)
- CPU handles:
    - Camera capture to pinned host memory
    - High-level scheduling / control
    - Consuming final poses (e.g. pushing to NetworkTables / Pose Estimator).

1.3 Success Criteria

- **Performance**:
  - Baseline: >40 FPS full-frame AprilTag detection at 720p input with decimated working resolution (e.g. 640×360).
  - Target: effective **120 Hz pose updates** using full-frame + ROI strategy.
- **Quality**:
  - Detection rate and pose accuracy comparable to apriltag3 CPU implementation at equivalent resolution.
- **Scalability**:
  - Architecture supports extension to up to 4 cameras with modest rework.
- **Robustness**:
  - Handles camera disconnects, GPU errors, and recovers gracefully.

1.4 Constraints

- Platform: **Jetson Orin NX**, using CUDA.
- Language: C++ (core) + Python wrapper (for dev/testing).
- Time budget: this is a non-trivial R&D project; we'll stage it in phases.

---

2. System Architecture Overview

2.1 Top-Level Data Flow

Per frame:

1. **Camera Capture (CPU)**
   IMX477 / AR0234 → raw frame into pinned host buffer (BAYER_RG8 / GRAY8).
2. **Upload & Preprocess (GPU)**

    - Async upload to GPU.
    - Fused kernel: raw → Gray8 → undistort → decimate → working image.
3. **AprilTag Detection (GPU)**
    - Kernel 1: gradients + edge maps.
    - Kernel 2: quad candidate extraction.
    - Kernel 3: decode + PnP.
4. **Results Return (CPU)**
    - Copy small detection list (IDs + poses) back to CPU.
    - Feed into robot code / NT.

2.2 Components

1. **Camera HAL (`ICameraDevice`)**
2. **Frame Manager (CPU)** – double/triple buffering of pinned host frames.
3. **GPU Context (`GpuContext`)** – streams + device buffers.
4. **Image Preprocessor (`ImagePreprocessor`)** – fused kernel(s) for raw→Gray.
5. **AprilTag GPU Detector (`AprilTagCudaDetector`)** – detection kernels.
6. **Scheduler/Pipeline (`CameraPipeline` / `MultiCameraSystem`)** – orchestrates frames.
7. **Persistent Kernel Framework (optional later)** – eliminates per-frame launch overhead.
8. **Python Wrapper (`c5b_apriltag`)** – for experimentation.

---

3. Component Design & Implementation Plan

3.1 Camera HAL & Frame Manager

Goal: Fast, predictable camera→CPU→GPU path with minimal overhead.

3.1.1 `ICameraDevice`

Already conceptually defined:

```
class ICameraDevice {
public:
    virtual ~ICameraDevice() = default;
    virtual const CameraStaticInfo&    staticInfo()    const = 0;
    virtual const CameraRuntimeConfig& runtimeConfig() const = 0;
    virtual const CameraCalibration&   calibration()   const = 0;

    virtual bool grabRaw(uint8_t* dst,
                         int dstStrideBytes,
                         RawFrameFormat& outFormat) = 0;
};
```

Implementation: `UvcCameraDevice` using V4L2/OpenCV.

- Configure for **1280×720** at desired FPS.
- Prefer `BAYER_RG8` or `GRAY8` for AR0234, `BAYER_RG8` or `BGR8` for IMX477.

3.1.2 Frame Manager (CPU)

Allocate **N pinned buffers** (2–3) using `cudaHostAlloc`:

```
struct HostFrameBuffer {
    uint8_t* ptr;
    int      stride;
};

class FrameManager {
public:
    FrameManager(int width, int height, PixelFormat fmt, int
numBuffers);
    ~FrameManager();
```

```
    HostFrameBuffer acquireForCapture(); // returns free buffer
    void releaseAfterGPU(HostFrameBuffer&); // after upload
};
```

Camera thread:

- `acquireForCapture()` → `grabRaw()` → enqueue for GPU.

3.2 GPU Context & Buffers

Goal: Preallocate all GPU memory and streams for predictable high FPS.

3.2.1 `GpuContext` extensions

```
class GpuContext {
public:
    explicit GpuContext(int deviceIndex = 0);
    ~GpuContext();

    cudaStream_t createStream(const std::string& name);
    void         destroyStream(cudaStream_t stream);

    RawGpuImage allocRawImage(int width, int height, PixelFormat fmt,
int numBuffers);
    GpuImage    allocGrayImage(int width, int height, int numBuffers);

    void uploadRawAsync(const HostFrameBuffer& host,
                        RawGpuImage& rawGpu,
                        int bufferIndex,
                        cudaStream_t stream);
};
```

- Allocate **rawGpu buffers** and **grayGpu buffers** in arrays to support double/triple buffering.

3.3 Image Preprocessor – Fused Kernel

Goal: One pass: raw → gray → undistort → decimate (→ optionally gradient precompute later).

3.3.1 API

```
struct PreprocessParams {
    bool  undistort;
    bool  decimate;
    float decimateFactor; // e.g. 2.0f
};

class ImagePreprocessor {
public:
    ImagePreprocessor(GpuContext& ctx,
                      const CameraCalibration& calib,
                      int inputWidth,
                      int inputHeight,
                      float decimateFactor);

    // Fused preprocess: rawGpu -> grayGpu
    void preprocessAsync(const RawGpuImage& rawGpu,
                         GpuImage& grayGpu,
                         const PreprocessParams& params,
                         cudaStream_t stream);

private:
    // Optional: precomputed undistortion map, stored on device
    GpuImage undistortMap_; // or a custom struct for float2 mapping
};
```

3.3.2 Implementation steps

1. **Startup:**
    - Precompute undistort map for 1280×720 (or working resolution) if `undistort = true`.
    - Store mapping from output pixel to input pixel (`float2`) in a device buffer.

2. **Kernel: `rawToGrayUndistortDecimateKernel`**
    - Grid: 2D, covering **output (decimated) resolution**, e.g. 640×360.
    - Per-thread (for each output pixel):
        - Compute or lookup undistorted input coordinates `(x_in, y_in)`.
        - For `BAYER_RG8`:
            - Sample Bayer pattern, compute intensity (e.g. average of nearby green pixels).
        - For `GRAY8`:
            - Direct sample.
        - Write Gray8 value to `grayGpu`.
    - Optimize for:
        - Coalesced reads from raw image.
        - Coalesced writes to gray image.
        - If using map, compute once on CPU/GPU and reuse.

Performance target: << 1 ms per frame at 640×360.

3.4 AprilTag GPU Detector – Kernels

We implement `AprilTagCudaDetector` as a sequence of optimized kernels over the decimated Gray image.

3.4.1 API

```
class AprilTagCudaDetector {
public:
    AprilTagCudaDetector(const AprilTagDetectorConfig& cfg,
                         const CameraCalibration& calib,
                         GpuContext& gpuCtx,
```

```
                          int workWidth,
                          int workHeight);

    std::vector<TagDetection3D> detect(const GpuImage& grayImg,
                                       cudaStream_t stream,
                                       const float cameraToRobot[16] =
nullptr);
};
```

Internally:

- Uses:
    - Device buffers for gradient/edge maps.
    - Device buffer for quad candidates.
    - Device buffer for final detections (TagDetection3D).

3.4.2 Kernel 1 – Gradient & Edge Map

Input: Gray8 image (decimated).
Output:

- Gradient magnitude (or binary edge map).
- Possibly orientation.

Kernel design:

- Block: 16×16 or 32×8 threads.
- Use shared memory tiles with halo for Sobel filter.
- Steps per pixel:
    - Load neighbor pixels into shared memory.
    - Compute Gx, Gy; magnitude = |Gx|+|Gy| or sqrt.

- Threshold to produce edge map.

3.4.3 Kernel 2 – Quad Candidate Extraction

Goal: Identify quadrilateral edge clusters as candidate tags.

Strategy:

- Divide image into tiles (e.g., 32×32).
- Each block:
  - Uses shared memory to analyze edge pixels in tile.
  - Builds line segments or short chains of edge pixels.
  - Groups segments into 4-sided shapes with geometric heuristics:
    - roughly rectangular,
    - roughly convex,
    - size above a minimum pixel area.
- Accepted candidates:
  - Append to a global `QuadCandidate` array using atomic add to maintain count.
  - Cap count to some max (e.g., 256 or 512) to bound worst-case.

Data structure:

```
struct QuadCandidate {
    float corners[4][2];  // x,y pixel positions
    float score;          // heuristic quality
};
```

3.4.4 Kernel 3 – Decode + PnP

One block per quad candidate:

- Normalize quad to tag coordinate frame (homography).
- Sample interior bit grid (e.g., 6×6 / 8×8), compute ID + Hamming.
- Reject if Hamming too high or margin too low.
- For valid tags:
  - Compute PnP (pose) on GPU:
    - Use known tag size + four corners.
    - Use simplified PnP (e.g., P3P + refinement or EPnP).
  - If `cameraToRobot` present:
    - Apply 4×4 transform to convert to robot frame.

Write results into a `TagDetection3D` array on device, with atomic increment of detection count.

Finally:

- Copy results to host (`cudaMemcpyAsync`) – very small data.

3.5 Persistent Kernel (Phase 2 / Advanced)

To approach 120 FPS, we want to **reduce per-frame launch overhead**.

3.5.1 Concept

- Launch a **single persistent kernel** with a small number of SMs reserved.
- Kernel loops over a work queue of frames:
  - For each frame: run preprocess + detect pipeline.
- CPU only:

    - Writes frame info (buffer indices) to queue in mapped memory.
    - Signals kernel via atomic flag.
    - Polls or waits for result.

3.5.2 Work Queue Structure

```
struct FrameJob {
    int rawIndex;
    int grayIndex;
    int jobId;
    int status; // 0=pending,1=processing,2=done
};

struct FrameQueue {
    FrameJob jobs[MAX_JOBS];
    int head;
    int tail;
};
```

Persistent kernel:

- Dequeues a job, runs full pipeline (preprocess + detection), marks job as done.

CPU:

- Enqueues a job after capture/upload, waits for `status==2`, then copies detections.

This is an advanced step; you can start with normal per-frame kernels, then replace them with a persistent kernel once everything works.

3.6 Multi-Rate / ROI Strategy (Practical Route to 120 Hz)

True full-frame heavy search at exactly 120 FPS may still be overkill. A practical architecture:

1. **Full-frame detection** at **30–40 FPS**, using the full pipeline described above.

2. **ROI-based detection** at **120 FPS**:
    - Maintain expected tag locations from last frame + robot motion.
    - For intermediate frames:
        - Only run detection kernels in small ROIs around these predicted tag positions.
    - That massively reduces per-frame workload.

Architecturally:

- `AprilTagCudaDetector` supports:
    - A “full-frame” mode (grid spans entire image).
    - An “ROI list” mode (kernel only processes bounding boxes).

---

4. `CameraPipeline` & Scheduler

4.1 `CameraPipeline` (Single Camera)

```
class CameraPipeline {
public:
    CameraPipeline(const CameraPipelineConfig& cfg, GpuContext&
gpuCtx);
    ~CameraPipeline();

    std::vector<TagDetection3D> processOnce(); //
non-persistent-kernel mode

private:
    GpuContext&           gpuCtx_;
    std::unique_ptr<ICameraDevice> camera_;
    FrameManager          frameMgr_;
    ImagePreprocessor     preproc_;
    AprilTagCudaDetector detector_;
```

```
    cudaStream_t         stream_;

    int currentBufferIndex_; // for double/triple buffering
};
```

Flow in `processOnce()`:

1. `HostFrameBuffer buf = frameMgr_.acquireForCapture();`
2. `camera_->grabRaw(buf.ptr, buf.stride, rawFormat);`
3. `gpuCtx_.uploadRawAsync(buf, rawGpu, idx, stream_);`
4. `preproc_.preprocessAsync(rawGpu, grayGpu, params, stream_);`
5. `detections = detector_.detect(grayGpu, stream_, cameraToRobot);`
6. `frameMgr_.releaseAfterGPU(buf);`
7. Return detections.

Later, for persistent kernel mode, `processOnce()` will instead:

- Post a job into the persistent kernel queue and wait for completion.

---

5. Python Wrapper for Dev

Expose a minimal high-level API:

```
import c5b_apriltag as c5b

gpu = c5b.GpuContext()
cfg = c5b.load_config("camera.yaml")
cam = c5b.Camera(gpu, cfg)

while True:
```

```
dets = cam.process_once()
for d in dets:
    print(d.id, d.pose)
```

Later add:

- Multi-camera support
- ROI mode
- Debug: visualize candidate edges/quads.